WF-Interop: Adaptive and Reflective REST Interfaces for Interoperability between Workflow Engines

Emad Heydari Beni, Bert Lagaisse, Wouter Joosen
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{emad.heydaribeni, bert.lagaisse, wouter.joosen}@cs.kuleuven.be

ABSTRACT

Software service providers are evolving towards a business process outsourcing (BPO) model to benefit from specialised services and facilities of external partners. Activation of external processes as well as having long-term and coarse-grained interaction with the outsourced processes results in remote workflow interactions between heterogeneous and federated workflow systems.

WF-Interop aims at addressing the interoperability issues by defining a set of REST interfaces that enable standardised communication between these workflow engines. The WF-Interop interface focuses on deployment, activation and progress monitoring of workflows. It intends to be an interface for new as well as the existing workflow engines in order to expose their functionalities in a RESTful architecture. Amongst all functionalities proposed by WF-Interop, some may not be supported by some engines. As such, our standard API should be adaptive to the capabilities of each workflow engine and be reflective to the consumers by describing supported capabilities on demand.

As a validation of the principles and architecture of WF-interop, we created a proof-of-concept middleware and prototyped an accounting workflow with outsourced billing workflow on top of it using jBPM[13] and Ruote[9] workflow engines.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; H.1 [**Models and Principles**]: Miscellaneous

General Terms

Design, Standardization

Keywords

Workflow Systems, Interoperability, Heterogeneity, REST, Adaptability , Reflectivity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ARM 2015, December 07-11, 2015, Vancouver, BC, Canada
Copyright 2015 ACM. ISBN 978-1-4503-3733-5/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2834965.2834966.

1. INTRODUCTION

A lot of online software service providers, i.e. application service providers (ASP) and software-as-a-service providers (SaaS), are evolving towards a business process outsourcing (BPO) model. BPO refers to the systematic and controlled delegation of many steps of a company's business processes to a specialized online software service provider.

For example, many companies are outsourcing the business process of invoicing their customers to specialized providers (see Figure 1). In the traditional approach, on-premise workflow systems to manage the billing process are using the online web services of a document management provider to create, layout, send and receive invoices to their customers. This billing workflow will have fine-grained, synchronous interactions with this document service to create, send, update and resend the invoices when these are not paid.

In a BPO context, this whole billing process is outsourced to a billing provider. The company consuming the service of the billing provider now only has a long-term, coarse-grained interaction with the billing process: start the billing process of a customer and notify me when the customer has paid. While such a long-term interaction is running for weeks, the company might ask or receive intermediate updates about the billing process, such as *bill sent*, *bill resent*, or *bill paid*.

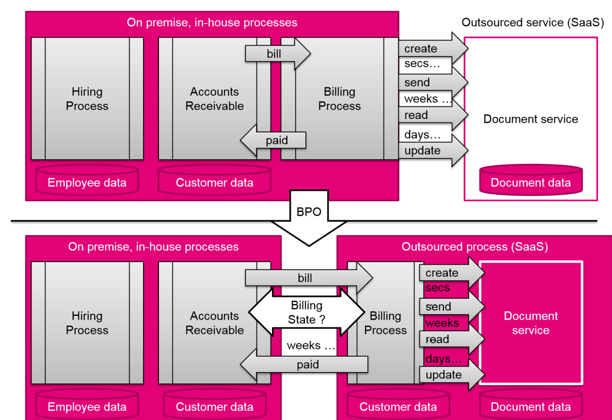

Figure 1: Outsourcing the billing process.

As illustrated in this example, business process outsourcing results in decentralized business process flows between companies and involves long term remote interactions between these work flows. Each of these companies typically has its own business process middleware (BPM) to sup-

port its automated business processes, such as jBPM[13], Ruote[9] or WWF[10]. BPO thus results in remote workflow interactions between heterogeneous and federated workflow systems.

Another example can be found in the design process of cars. Engineers follow a predefined business process when designing a part of a car (e.g. a door). This business process is typically a series of human tasks such as designing a 3D model of a door, then applying a mesh structure to it, etc. Now and then, the engineer wants to evaluate his design, e.g. by simulating the influence of vibrations or noise. This requires a remote simulation workflow with specialized scientific algorithms, offered as a service by a specialized company. Such a simulation can take weeks. Hence, the engineer would like to get updates about the progress of the simulation, e.g. in terms of steps that are completed or just in terms of percentage.

In each of these applications, a similar pattern emerges:

1. There are two federated parties (e.g. two companies) with heterogeneous workflow technologies (e.g. a jBPM workflow engine and a workflow engine for scientific simulations).

2. The first party starts a remote workflow on the second party's workflow system, and awaits its result. This is typically a long term process that can take weeks or months.

3. The first party wants to follow up on the progress now and then. This inspection interaction can be a pull request by the first party or a push notification by the second party. The progress inspection is typically decoupled from the internal task implementation of the remote workflow and expressed in a higher abstraction, e.g. *payment pending* or *simulation 65% completed*.

However, this similar pattern has to be implemented over and over again in each application, and for each type of workflow. Current workflow middleware only coordinates and supports short-term remote interactions such as synchronous or asynchronous calls to web services via SOAP or REST, on top of which the application-specific long-term interaction needs to be implemented.

Such a *frequently occurring, long-term remote interaction between heterogenous workflow technologies* requires (screams for) middleware support, on which workflow applications can leverage to reuse the long-term remote interaction pattern, including intermediate inspection via push or pull.

In this paper we introduce WF-interop. WF-Interop defines a set of interfaces that enable standardized communication between federated and heterogeneous workflow engines. The WF-Interop API focusses on deployment, activation and progress monitoring of workflows.

The WF-interop interfaces define both a reflective and adaptive middleware for workflows.

- Depending on the workflow engine, and the actual workflow, the interface of the middleware adapts and publishes the supported operations at that moment. (e.g. the pause operation for a workflow is only exposed if the workflow engine supports it and if a workflow is actually running.)

- Running workflows can be inspected in terms of higher application-specific abstractions, depending on the type of workflow. WF-interop supports both a pull and a push model for state inspection.

Next to these advanced features, WF-interop also defines basic operations for workflow engines, such as:

- Workflow engines can be inspected for which types of workflows a specific engine supports.

- Workflow engines can be asked to create new workflow types (via the deployment interface).

- Workflow engines can be contacted to instantiate new workflow instances (via the activation interface).

The contribution of WF-interop is threefold. First, WF-interop introduces a standardized management interface for workflow engines to enable interoperability. Second, it introduces a reflective and adaptive approach to support discoverability, evolvability and adaptability of engine management interfaces. Third, to increase industry adoption, the WF-interop interfaces are currently supported in a REST-based architecture between workflow engines, but also leverage well-known principles such as *Hypermedia as the Engine of Application State* (HATEOAS) as a novel technique to support adaptive management interfaces in middleware.

The rest of this paper is structured as follows. Section 2 describes the main principles and the rationale behind WF-Interop interfaces. Section 3 introduces a set of RESTful interfaces with a particular attention to HATEOAS constraint of REST architecture. Section 4 validates the aforementioned concepts in a middleware and a prototype application. Section 5 compares our approach with related work and existing systems. Section 6 concludes this paper and outlines our research outcomes.

2. WF-INTEROP PRINCIPLES

WF-Interop aims to standardize the communication between federated and heterogeneous workflow engines with an adaptive and reflective approach to enable discoverability and evolvability of the management interfaces.

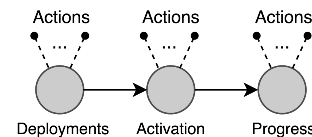

Figure 2: Sequence of resources and their actions.

WF-Interop promises *self-discoverability* of functionalities for consumers without the necessity of having static documentation. It assumes that workflow engines contain resources such as *deployments*, *activations* and progress *monitoring* as shown in figure 2. Various actions can be executed on these resources which may bring the execution flow as well as the internal states of the enactment engine to a different state. Upon execution of each action on a resource, WF-Interop guides consumers by proposing next possible moves in the form of resources and their actions based on the current application state. Consumers are able to read a short on-demand description for each of the proposed actions in order to figure out the protocol, input properties and a brief description of the semantics of the action. They may

traverse the application state by invoking these proposed actions on the same or the other resources.

This mechanism introduces API *evolvability* for workflow engine providers in terms of new functionalities on top of WF-Interop. WF-Interop can be adopted as a base interface. Later on, new resources along with their necessary actions can be defined and added to the appropriate application states. The newly added resources are discoverable by the navigational information sent back to the consumer upon each action invocation. Workflow engines come with different capabilities; some may not cover all minimum functionalities defined by WF-Interop. The aforementioned mechanism enables all providers with different levels of support to be *adaptive* to the current state of their production.

3. WF-INTEROP REST ARCHITECTURE

In this section, we propose a RESTful interface, called *WF-Interop*, to standardize the communication between federated and heterogeneous workflow engines. In addition, we transform our proposed interface to an adaptive and reflective solution by using *Hypermedia as the Engine of Application State*, known as *HATEOAS*.

WF-Interop focuses on four fundamental aspects of workflow engine interactions: (1) deployment of workflow definitions, (2) activation and (3) progress monitoring of process instances within workflow systems including (4) observers.

According to our study based on runtime interfaces of several workflow engines and Workflow Management Coalition (WfMc) standards, the workflow definition deployment and its related functions are the first necessary requisite in our RESTful api proposal. In the first draft of the deployment interface as listed in table 1, one can deploy, undeploy, modify, delete, and fetch workflow definitions.

The second interface provides activation functionalities for workflow process instances. The fundamental methods for these resources are listed in the table 2. It includes functionalities such as instantiation, aborting, pausing and resuming of process instances.

The last interface, specified in table 3, provides progress monitoring for workflow process instances. This interface standardizes the interactions between enactment engines to support progress reporting in a push and a pull model.

WF-Interop, as described in the previous paragraphs, aims at addressing interoperability issues of workflow engines in a RESTful architecture. The rationale behind REST architectures is described in Roy Fielding's dissertation [4] and the fourth layer of the Richardson Maturity model [5]. One of the principles in REST is to utilize *Hypermedia as the Engine of Application State* (HATEOAS). HATEOAS is described as a *constraint* of REST and supports the aforementioned architectural features proposed by WF-Interop such as *self-discoverability*, *evolvability* and *adaptability*.

As illustrated in figure 3, WF-Interop resources are dependent on each other and each resource has a set of actions. Relying on *Hypermedia*[12] based principles, one is able to start from one of these resources and explore the remainder of the dependency graph based on the current state of the application. In other words, consumers of the workflow engine are supposed to receive a list of possible actions after execution of an action. These proposed actions come with on-demand documentations enabling the clients to navigate the API's without any prior knowledge about interaction with workflow engines. For instance, one may execute a `GET`

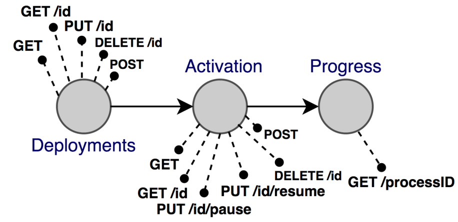

Figure 3: RESTful WF-Interop

method on the `Activation` resource in order to fetch a long running business process. Depending on the current application state of the process and the policies of the enactment engine, WF-Interop may respond back some other actions in Hypermedia format such as links to methods of the progress monitoring resource for more comprehensive reporting or links to obtain intermediate results. All of the suggested links are based on each engine's capabilities which makes the RESTful WF-Interop *adaptive* to all engines.

This architecture is *change tolerant* in the sense that workflow engines are able to either implement new functionalities or even deprecate some actions by manipulation of hypermedia links propositions. For example, some engines can implement some extra methods on top of the Progress resource for additional capabilities and add those actions as links to the response bodies upon proper Activation method calls.

4. VALIDATION

As a validation of the principles and architecture of WF-interop, we created a proof-of-concept middleware and, on top of it, prototyped an accounting workflow with outsourced billing workflow. The goal of this validation is to illustrate the adaptability and the reusability of the different layers in the middleware to support long-term remote interactions between heterogeneous workflow technologies.

We first provide more details about the accounting workflow and the different workflow technologies. Then we discuss the layered architecture.

In one of the activities of the accounting business process, the host workflow engine outsources the billing process to another engine. Since the billing process might be a long running process, the accounting process typically has a coarse grained interaction with the billing process at the application level (the top-level workflow). However, the accounting process will get intermediate on-demand progress updates based on the current state of the process instance running on remote workflow engine. This progress information is about the shipment (e.g. bill-sent, bill-resent, etc.) and the current state of the payment (e.g. paid, not-yet-paid, etc.). Upon termination of the remote billing process, the results become available to the accounting process and it continues the rest of its activities.

We used the jBPM workflow engine for the accounting process and the Ruote workflow engine for the billing process in order to have heterogeneous engines. Besides, all of the interactions are done using WF-Interop interfaces. In figure 4, the four-layer architecture of the prototype is illustrated.

Application layer. In the top layer (4), i.e. the application layer, the accounting process is defined with all of its relevant activities. As shown in figure 5, the first and

Table 1: Workflow Deployment Resources

Method	URI	
GET	`/deployments/`	Get all Workflow definitions
POST	`/deployments/`	Deploy a Workflow definition
GET	`/deployments/{id}`	Get a Workflow definition
PUT	`/deployments/{id}`	Update a Workflow definition
DELETE	`/deployments/{id}`	Undeploy a Workflow definition

Table 2: Workflow Activation Resources

Method	URI	
GET	`/activations/`	Get all workflow process instances
POST	`/activations/`	Start a workflow process instance
GET	`/activations/{id}`	Get a workflow process instance
DELETE	`/activations/{id}`	Abort a workflow process instance
PUT	`/activations/{id}/pause`	Pause a workflow process instance
PUT	`/activations/{id}/resume`	Resume a workflow process instance

Table 3: Workflow Progress Resources

Method	URI	
GET	`/progress/{processInstanceID}`	Get the current state of a workflow process instance
GET	`/progress/{processInstanceID}/observers/`	Get the list of observers
POST	`/progress/{processInstanceID}/observers/`	Subscribe an observer
DELETE	`/progress/{processInstanceID}/observers/{observerID}`	Unsubscribe an observer

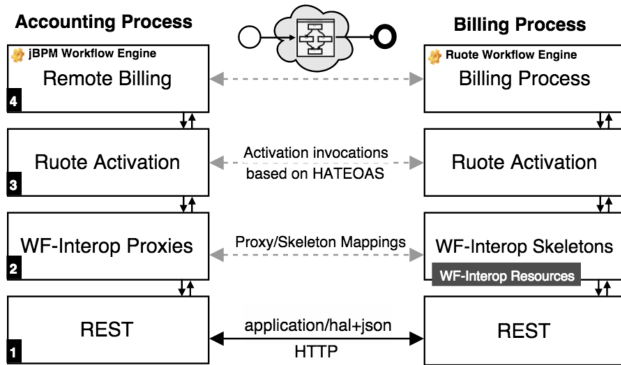

Figure 4: Outsourcing the billing process.

second activities are Human Tasks[11] responsible for user input such as selection of required steps in the billing process and billing information. The last activity, shown as Remote Billing, is responsible for outsourcing the billing process using the acquired input arguments. At this level, there is no dependency to WF-Interop and the complexity of interface discoverability, adaptability and intermediate progress updates is encapsulated in the (*reusable*) middleware layers underneath.

Engine layer. The third layer, i.e. the engine layer, is the top-level middleware layer that offers an abstraction towards the application layer to execute remote workflows as a single, long-term interaction. This layer is application-independent (no concept of accounting or billing), but it is technology-specific. The abstraction that is offered is specific to execute remote Ruote workflows from a jBPM engine.

This abstraction supports the high level execution of Ruote engine instructions on top of WF-Interop. The abstraction itself depends on the *activation* and *progress* resources in WF-interop, and has the form of a *reusable* sub-process[11] in jBPM. This sub-process will be executed as a child process of the Remote Billing activity in the application layer. The sub-process contains an algorithm to handle all the logic with regard to progress updates as well as discoverability and adaptivity of the remote engine's interface.

As such, the engine layer is decoupled from the accounting logic provided in the application layer and encapsulates the algorithm as a *reusable* sub-process for managing remote Ruote engines and Ruote process instances within the jBPM engine. As shown in figure 6, all of the interactions are based on HATEOAS navigational information upon execution of each instruction.

Next, we illustrate this algorithm for instantiating a remote workflow asynchronously. When a `Create-Process` is executed, the actual instantiation of the process is asynchronous. Hence, as long as the process is not started completely in the Ruote engine (i.e. asynchronous instantiation), the proposed instructions are limited to `Get` and `Abort`. As soon as the process instance is effectively started, there are more instructions available that can be executed in this current state of the process instance (e.g. `Pause`). As discussed in previous sections, this mechanism makes the interactions adaptable to the capabilities of workflow engines at different process states.

This algorithm is implemented as a jBPM process and executed as a sub-process of the parent process in the application layer. A simplified version of the sub-process is illustrated in figure 7.

Basic WF-interop layer. In the second layer, the basic WF-Interop resources and their associated operations are supported. This layer contains reusable middleware support in the form of WF-interop proxy objects. WF-Interop proxies hold information like process definition identification, process instance identification, a generic key-value dictionary for process variables, etc. Each of the WF-Interop activities, as shown in figure 7, is implemented as a jBPM activity in the form of a custom work item, and the native jBPM data is mapped to the WF-Interop proxies. These

Figure 5: Layer 4 - Accounting Process including a Billing Business Process Outsourcing.

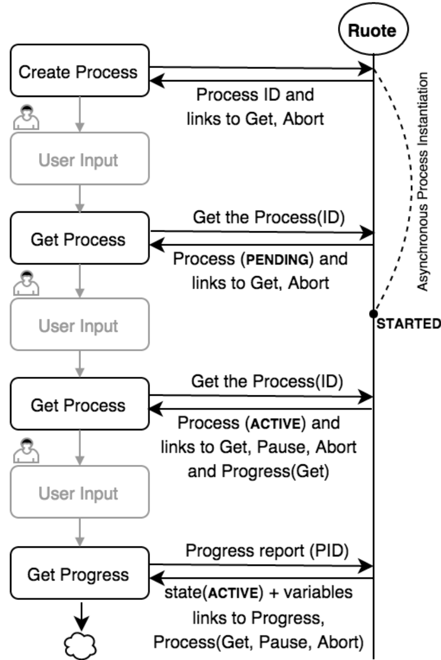

Figure 6: Layer 3 - Process creation and inspection

custom work items can be *reused* in order to create different layers on top, such as our engine layer. This layer is also responsible for JSON[3]/XML marshalling and unmarshalling, for example, when a progress report is received from the Ruote engine. After unmarshaling the data received from Ruote, HATEOAS based instructions in the form of links are obtained.

REST Transport layer. The first layer, i.e. the bottom transport layer, is using standard REST technology to transport the engine instructions and related data. More specifically, this layer sends instructions and data with content type `application/hal+json`[8] over the HTTP protocol in a RESTful service architecture.

Content type `application/hal+json` is an extension to the `application/json`[2] content type that supports some extra fields representing further possible instructions in the form of hypermedia links[12]. As shown in code listing 1, this is how a response looks like when one creates a process in the Ruote engine via WF-Interop. The `process:get` and `process:abort` are custom link relations and the `href` is pointing to the proposed resource. Each of the hypermedia links comes with an on-demand execution and semantic description. In order to fetch each documentation, the `curies`[1] syntax has been used. In other words, if one wants to invoke the `abort` instruction, the `process:abort` relation should be used. Firstly, the resource URI can be obtained from the `href` field of the link. Afterwards, by inspecting the proper `curie` regarding the `abort` relation, some necessary information becomes available such as the semantics of the `abort` function and the associated engine behaviour, input arguments like process instance identification, etc. As listed in table 4, several resources are defined within WF-Interop for this purpose.

Listing 1: POST /activations/

```
{ ...
  "_links": {
     "process:get": {
        "title":"Fetch the process",
        "href":"/activations/1234" },
     "process:abort": {
        "title":"Abort the process",
        "href":"/activations/1234" },
     "curies": [{
              "href": "<process-doc-url>/{rel}",
              "name": "process",
              "templated": true
 }]}}
```

Table 4: Workflow Relation Resources

Method	URI	
GET	`/rels/`	Get all relation types
GET	`/rels/deployment/{rel}`	Get a relation description of the deployment type
GET	`/rels/process/{rel}`	Get a relation description of the process type
GET	`/rels/progress/{rel}`	Get a relation description of the progress type

5. RELATED WORK

The Workflow Management Coalition (WfMc) published an `XML` based protocol called *Wf-XML*[14] which is an extension of the ASAP[6] protocol for runtime integration of process engines. WF-XML enables the consumers to have a standardized communication with workflow engines by outlining a set of pre-defined instructions based on `SOAP` messages. The properties of each instruction are opened up to the consumer by `get-properties` instructions, but instructions themselves are not discoverable. Hence, it is not adaptable to the engine-specific capabilities.

In a study of B2B business process integration[7] , the authors recommend three approaches for process integration: (1) workflow system interoperability (e.g. Wf-XML), (2) web service choreography and (3) a multi-phase process composition approach. The recommended approach is the latter which is a combination of the former ones. This study is also based on the `SOAP` protocol. This approach has some level of controllability by introducing public and private processes, but still does not address adaptability and change tolerance to the workflow engine's API.

In the SHIWA[15] project, the interoperability of scientific workflow systems gets addressed using a coarse grained

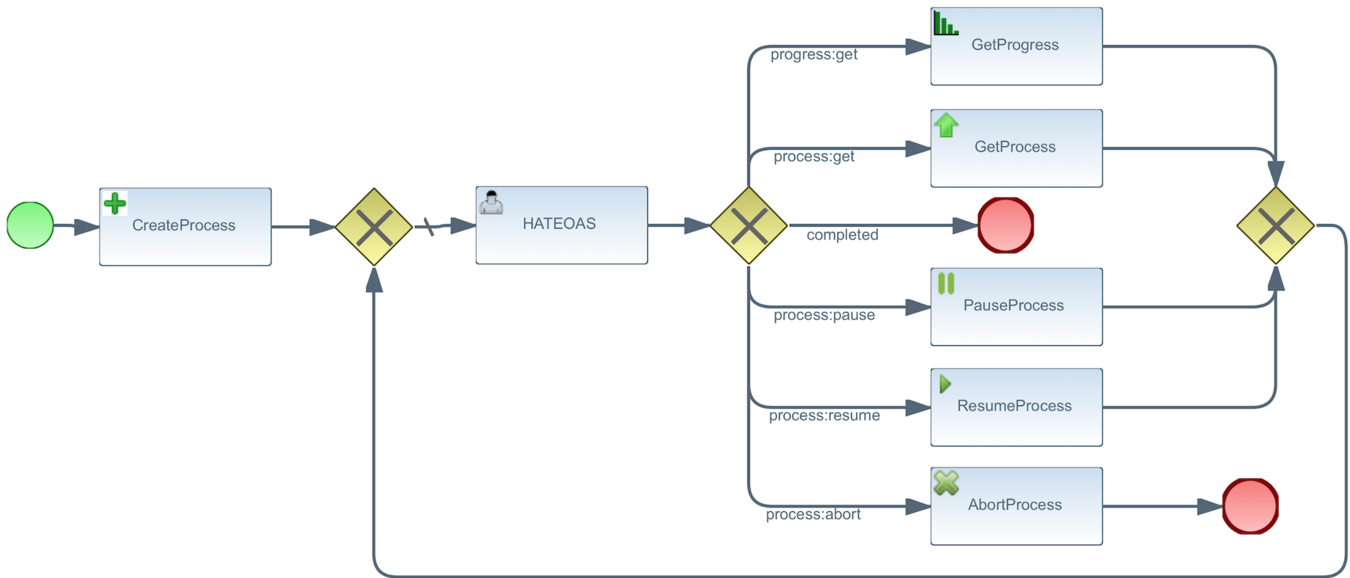

Figure 7: Layer 4 - Ruote Activation

platform by hiding engines and representing workflow abstracts instead. Hiding engines in the context of scientific workflows based on this platform might be acceptable, but in business process outsourcing may not. In addition, if any of the underlying engines does changes in its API's, it needs further development at the platform side to be adaptive to the recent changes. In other words, engines are not decoupled from the platform itself.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced WF-Interop which enables standardized communication between federated and heterogeneous workflow engines. It enables workflow engines to outsource workflow processes to external parties and monitor their progress. WF-Interop defines RESTful interfaces for inter-engine communication and supports discoverability and evolvability of an engine's interface. WF-interop leverages *HATEOAS* (*Hypermedia as the Engine of Application State*), a well-known constraint of Fielding's REST architecture. This mechanism makes WF-Interop adaptable to the capabilities of many engines (e.g. jBPM and Ruote) and supports that engines evolve with new features.

We validated WF-interop in a proof-of-concept middleware and illustrated its use with an accounting workflow that outsources the billing workflow between a jBPM and Ruote workflow engine. Both engines remained unchanged while all WF-Interop middleware support could be modularised in *reusable* software assets such as application independent sub-processes and workflow activities. Our future work includes the further validation of WF-interop in two industry applications - as introduced in the introduction of this paper.

7. REFERENCES

[1] M. Birbeck and S. McCarron. Curie syntax 1.0. *W3C Candidate Recommendation CR-curie-20090116, January*, 2009.

[2] D. Crockford. The application/json media type for javascript object notation (json). 2006.

[3] S. Ecma. Ecma-262 ecmascript language specification, 2009.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[5] M. Fowler. Richardson maturity model: Steps toward the glory of rest. 2010.

[6] J. Fuller, M. Krishnan, K. Swenson, and J. Ricker. Oasis asynchronous service access protocol (asap), 2005.

[7] J.-Y. Jung, H. Kim, and S.-H. Kang. Standards-based approaches to b2b workflow integration. *Comput. Ind. Eng.*, 51(2):321–334, Oct. 2006.

[8] M. Kelly. Json hypertext application language. 2013.

[9] J. Mettraux, K. Kalmer, R. Meyers, H. de Mik, A. Kohlbecker, M. Barnaba, G. Neskovic, N. Stults, O. Pudeyev, M. Gfeller, et al. Ruote-a ruby workflow engine.

[10] M. Milner. A developer's introduction to windows workflow foundation (wf) in. net 4. *Retrieved from: on Oct*, 11(2010):47, 2009.

[11] B. P. Model. Notation (bpmn) version 2.0. *OMG Specification, Object Management Group*, 2011.

[12] M. Nottingham. Rfc5988: Web linking. *Internet Engineering Task Force (IETF) Request for Comments*, 2010.

[13] RedHat. jbpm business process management suite.

[14] K. D. Swenson, S. Pradhan, M. D. Gilger, M. Zukowski, and P. Cappelaere. Wf-xml 2.0 xml based protocol for run-time integration of process engines. *Workflow Management Coalition*, 2004.

[15] G. Terstyanszky, T. Kukla, T. Kiss, P. Kacsuk, Á. Balaskó, and Z. Farkas. Enabling scientific workflow sharing through coarse-grained interoperability. *Future Generation Computer Systems*, 37:46–59, 2014.